Towards functional programming in LATIN2

Moritz Blöcher

Supervised by
Dr. Florian Rabe

at
Friedrich-Alexander-Universität Erlangen-Nürnberg

June 2,2022

Abstract

Programming languages are the framework programs are built in. For knowledge representation of programs a framework for such languages is needed. Latin2 provides a basic mathematics and logical library for knowledge representation, which now is extended towards programming languages. The functional programming languages are the most similar to a logical representation in lambda terms and thus, concepts needed for such languages are developed first, followed by a discussion on some implementation details and mutable features.

Declaration

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen 2.6.2022 Moritz Blöcher

Contents

1 Introduction

Motivation Knowledge representation is an important task. There was success in building logical and mathematical frameworks to represent the knowledge in those domains. Obviously, a large area for computer science are algorithms and their implementation as programs. These programs are realized in different programming languages. For knowledge representation it would be useful to build a framework which can represent all the programming languages. This can be done through a modular approach to the types and concepts of programming languages. It would be beneficial if the formalization not only allows for representation but also for execution of representative programs. Functional programming languages are closest to logical representation and thus are the easiest group to begin with.

Contribution This work describes the data types formalized for a basic functional programming language and also discusses different possibilities of realizing concepts used in programming languages in general. The work extends the number theories and makes them feasible for programs. It introduces strings to Latin and formalizes list-like data types as collection types. The work adds specific monad types to the already existing monad structure. The thesis also proposes future changes to MMT needed for a real environment for programming.

Overview In chapter 2 related work is described. In chapter 3 MMT/LF is briefly explained and and an overview of Latin is given. Chapter 4 details the formalized basic data types, while chapter 5 shows the realization of more complex data structures using the monad structure. Chapter 6 describes the concepts of recursion in MMT and error handling techniques. This chapter also gives an outlook on functions which use mutable data types. The last chapter concludes the described work and outlines future work.

2 Related Work

This work concentrated on formalization of data types in functional programming. It was inspired by libraries of functional programming languages with various degrees of emphasis on formalization.

SML Basis Library Standard ML Basis Library is a standardization of ML languages. The library provides an interface for basic data types. It provides simple support data types like list and option. It does not support higher data types. The library only specifies the data types and some basic functions operating on them. A concrete implementation is not given. [2] [7]

Haskell Haskell is a popular and widely used functional programming language. The language has integrated monads to a often used tool kit. The programming language provides the basis of monadic programming with the classes of *Functor*, *Applicative*, and *Monad*. The different instances of specific monad types like *IO*, *Maybe*, and *State* then realize the framework specific for their type. The types provide a functional solution for mostly non-functional problems. [3] [1]

3 Preliminaries

3.1 MMT/LF

The Meta Meta Tool (MMT) is being developed at FAU Erlangen Nürnberg. This tool is capable of designing logical frameworks as well as working with them [5]. MMT already provides simple logical foundations, of which MMT/LF is the most popular one. It formalizes the Edinburgh Logical Framework in MMT. In this framework Latin is developed.
MMT documents consist mainly of *theories*, especially those described in this paper. These consist of a list of constant declarations, which can contain a type declaration, definition, notation declaration, and a role. A simplified informal syntax of a *theory* is described in figure 1.

Theory	:=	`theory` Name [: Metatheory] `=` Body
Name	:=	string
Metatheory	:=	*URI*
Body	:=	(Theory | Include | Constant)*
Include	:=	`include` *URI*
Constant	:=	Name : *Type* [`=` *Definition*] [`#`*Notation*] [`role` *Role*]

Figure 1: Informal syntax of a theory

Additionally MMT supports the morphisms *include*, *view*, and *structure*. If *theory* T needs to use already defined declarations of Theory S it can include this theory through its URI.
The *view* morphism is the realization of one theory through another. In the view all declaration of its domain need to be expressed through elements of the codomain. In the view all instances have to realized and all axioms of the interface need to be proven. The morphism *structure* is similar to view. It is in simple terms a view from one theory to itself. This morphism will automatically realize all not mentioned declarations. Using *structure* allows for renaming of functions and types but essentially creates a distinct new theory with the same properties.

MMT allows to connect formalized content and programs. A type defined in MMT can be matched to one defined in Scala. In the MMT code a variety of types and functions operating on them are predefined. These can be connected using the rule *rule* like in figure 2. In this example, the elements $nat/Number$ and $zero$ are the types in Latin, while $StandartNat$ and $Zero$ are the corresponding Scala expression given through their path in MMT.

LF LF (Edinburgh Logical Framework) is used as a meta-language for the formalization of deductive systems [6]. LF is a simplistic dependently typed lambda calculus [10]. Due to its reduced syntax set LF is a relatively weak logical framework which is suitable as a meta-language. The notation for the relevant operators

```
rule rules?Realize nat/Number uom?StandardNat|
rule rules?Realize zero uom?Arithmetic/Zero|
```

Figure 2: Example for rules in natural numbers

for formalization is shown in Figure 3. For Latin separate types are declared. The two important concepts are *tp* (type) and *tm* (term).

```
theory LambdaPi =
   include ?Kinded|
   Pi     # { V1T,… }
   lambda # [ V1T,… ]
   apply  # 1 %w…
   arrow  # 1 →…
|
```

Figure 3: Core elements of LF syntax

3.2 Latin

The Logic atlas and integrator (Latin) is a project which focuses on building a framework for knowledge representation. This framework tries to represent meta theoretic foundations of mathematics and logic as well as to connect these foundations on a meta logical level. The project provides a library for MMT called Latin2. This library consists of an extensive amount of theories in diverse fields of logical and mathematical theories. The library provides theories of for example propositional logic, higher order logic, and model logic for logical representation. It also includes mathematical theories like set-theory, topology, category-theory, and type-theory. With this library the project provides a platform for representing knowledge from different disciplines and tools. [8] [9]

4 Types

A core element of programming languages are their types. They build the foundation the functions operate on. Programming languages provide a base set of types which can be extended through a modular library. It allows the users to introduce the complex structures in the form they need. Each programming language provides only its fixed base set of types, which is typically not compatible to other languages. The advantage of such a fixed set is a better optimization of the core elements and thus better performance.
In this work it is tried to formalize the types in Latin2 for a functional programming language as modular as possible so that different realizations of the same types can coexist and be used with the now formalized theories. The modularity additionally allows for representation of programming languages with different foundations.

These types can now be formalized in different depths. The easiest form is like it is done in the SML Basic Library. There, only the types are declared and the signatures of the functions operating on them are specified. This form of formalization realizes an interface for real programming languages. A more in-depth realization uses additional axioms defining the functionality of the given functions and giving the underlying structure of the types. The most in-depth formalization also adds the realization of the functions.
The formalization approach in this work mostly consists of a specification of functions and types. Additional axioms are defined for these functions for clarification of functionality. These axioms are used as rewriting rules for execution purposes when possible. In the following the base types Boolean, Numbers, and String as well as the collection types are described.

4.1 Base types

Boolean There are two variants a truth value can be implemented. One variant is to formalize it as *tp*, the type representing type in Latin. This has the advantage that all other types the language will formalize are of the same type.
The other variant is to formalize the boolean type on a lower theoretical level. This type is already formalized as $type_{LF}$ and is called *prop*. There are problems but also clear advantages of this second variant. Latin2 uses *prop* already extensively and provides truth to a large set of logical frameworks. With the use of *prop* these theories could then be utilized in the programming languages. A problem of *prop* is that the type *tm* representing terms is formalized in Latin as $tp \to type_{LF}$. Therefore *prop* can not be a *tm*, but most functions will have $tm\ a$ as argument and thus can not use *prop* as a. The boolean specific functions can be properly formalized but in general still need a boolean *tp* to mitigate the problem. Through transformation functions boolean can then be turned to *prop*. The *prop* also has the benefit of being better supported by MMT because of its wide use.
Do to the benefits of *prop* this variant is the preferred one in formalization. When this is not possible, *bool* of theory Booleans is used.

lazy logic Functional programming often uses lazy evaluation. In programming generally lazy logic is an important part of conditional statements, where only evaluating necessary parts of the formula and terminating as early as possible can be crucial. The already existing logical connectives in Latin were only used for formulas and don't need these properties. The new logical connectives no longer use the usual form of formalization but use simplification. Through this change the formula is evaluated one part at a time and only uses the next part if the result is not already clear.

Numbers Numbers are an essential part of programming. Most languages contain an implementation through fixed precision numbers only allowing for a fixed amount of digits. They mostly provide floating point and integer numbers in different degrees of precision. Typically these integers don't distinguish between the already existing mathematical number sets (natural and hole numbers), while the floating point implementations only represent a part of the real number spectrum.

Most languages also contain more complex forms of these representation in the libraries not relying on a fixed amount of digits. This resolves some of the problems the number types have but is still no accurate representation of math.

Latin2 models mathematical knowledge, thus the number system was implemented through the mathematical number sets. Since the different sets of numbers are subsets of each other they can be built incrementally and thus can use the already formalized functions available from the subsets. However, the validity of certain theorems only hold in their respective set of numbers. This problem can be handled through separating these theorems in specific theories not included in the supersets.

The described system can model the mathematical number theory, but only works correctly if one number type is used at a time. The problem is that if more number sets are included the largest type would be used. To overcome that problem the specific numbers are realized through *structure* in MMT. This approach results in completely different structures for the types. While this is a good solution for the described problem it also has its own flaws. The drawback of this separation is the difficulty of implementing a version of sub typing. The implemented version of numbers thus show the problem that numbers of different types aren't compatible with each other. For example an integer 2 can't be used for addition with a 2 of the real numbers. These problems could probably be fixed with a better knowledge of sub typing and more time.

The number types are built up incrementally starting with natural numbers followed by integer and rational numbers. The natural numbers contain the basic type of numbers and the constructors to build them are zero and successor. Additionally, they provide simple functions, such as addition and multiplication. This theory also contain some other simple relations. The integer theory adds the negation function for negative numbers, predecessor, and subtraction. As a result of negative numbers, this theory contains specialized computation rules for addition and multiplication. The real numbers add division into the theories.

The representation of numbers trough succ and zero is practical for formalization of the functions operating on them. But a programming language also works with instances of numbers and the representation of them in this form is unpractical and inefficient. Through the use of rules in MMT it is possible to provide a more programmer-near form of representation. Through rules a formalized fragment can be connected to code written in Scala. In the case of numbers the numbers themselves and some simple functions are implemented in Scala. This additional formalization enables to use normal numerals in MMT. An additional consequence of this is that calculations mapped to Scala are performed as real programs instead of by rewriting.

String Strings are a base type of programming. They are the realization of a chain of characters. The implementation of this type is done either using a list or an array of chars. The here formalized type does not define the type structurally and does not need to decide on the specific realization. String is formalized simply

as *type* without structural definitions. Nevertheless formalization of the functions of String need a structure they operate on. This structure is given by the constructors and destructors *string_to_char* and *char_to_string*. These functions are translating between the String type and the structure of list of chars. These functions need to translate between the instances of strings in Scala and the formalized structure in MMT. This can be done solely in Scala translating between representation of string and the list of chars.
The string is realized in this way because operating on instances is more efficient in Scala. The storage of strings should not be relevant for formalization. Because then different structures can be designed by adding new constructors and destructors. Also the storage of strings can be adjusted without the need of changing the formalization of the functions.

In addition to the base concept of *String* some simple functions are formalized with the framework described before. A summary of these functions is given in figure 4.

functions
string_to_char
char_to_string
string_concat
size
charAt
substring
equal

Figure 4: String functions

In addition to the already mentioned types, Latin contains types which don't need to be changed for use in the functional programming languages. One class of these types are the product types. Of these the *SimpleProduct* realizes tuples. The formalization is shown in the figure 5.

```
theory SimpleProducts =
  include ?SimpleProductTypes❙
  include ?TypedEquality❙
  simppair: {A,B} tm A → tm B → tm A ×▯ B|# 3 ,▯ 4 prec 50❙
  simppi1 : {A,B} tm A ×▯ B → tm A|# 3 ₁▯ prec 60❙
  simppi2 : {A,B} tm A ×▯ B → tm B|# 3 ₂▯ prec 60❙

  compute1 : {A,B,a:tm A,b:tm B} ⊦ (a ,▯ b)₁▯ =▯ a|role Simplify❙
  compute2 : {A,B,a:tm A,b:tm B} ⊦ (a ,▯ b)₂▯ =▯ b|role Simplify❙
❚
```

Figure 5: SimpleProducts theory

4.2 Collection Types

There are two essential theories already formalized which are essential for the collection types: EndoFunctor and EndoMagmas. In EndoFunctor the applied type is defined, it is a type of types. In practice this allows to construct complex types building a structure around an arbitrary type. The EndoFunctor also formalizes a map function. Building upon EndoFunctor, the EndoMagma theory contains the basic operator *op* which connects two applied types to one. Additionally it describes different basic mathematical frameworks, for example EndoCommutative specifies the commutative law for *op*, while EndoNeutral adds the neutral element.

The Collection type now uses these theories to build a framework for specific data types. Collection adds two simple functions: cons and singleton. Singleton allows to create from an object the collection version of that type. Cons adds an object to an already existing collection. These two functions allow to build list-like data structures commonly used in programming languages.

The Collection type also contains a neutral element. This element is defined in the EndoNeutral theory. This neutral element is formalized on a variable type whose type is inferable. This is not possible if no type is used in the term. The function null_collection solves this issue enabling MMT to easily derive the right type.

The formalized Collection type not only contains the definition of the structure and the constructors but it also provides functions operating on the structure. The formalized functions are all built in the same way. First the head of the function is declared, then the functionality is defined. The head of the function contains the name of the theory, the type declaration, and the way the function will be used. The specification of functionality is done by axioms. For the Collection type each function specifies at least one axiom per constructor neutral, singleton, cons, and op (Figure 6). The axioms are not written in a specified logic but in LF directly, which is different form axioms formalized in more abstract theories in Latin. The advantage of quantifying over the elements in first order logic is the ability to reason in that logical theory. On the other hand, by using the *kind* syntax of LF for quantification the reasoning happens in higher logic. The advantage of axioms as equality in LF is the easy use of simplification rules of MMT. These enable the execution of the function through the axioms. It is not strictly necessary to provide axioms for all constructors, a smaller sett would suffice for axiomatic description of the functionality. But all four constructors have their use in formalization and thus need their axioms. In complex functions it is not always sufficient to only have one axiom per constructor. In that case, the name indicates the case it represents.

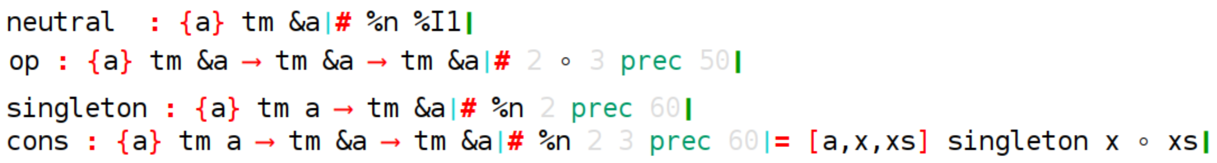

```
neutral  : {a} tm &a|# %n %I1|
op : {a} tm &a → tm &a → tm &a|# 2 ∘ 3 prec 50|
singleton : {a} tm a → tm &a|# %n 2 prec 60|
cons : {a} tm a → tm &a → tm &a|# %n 2 3 prec 60|= [a,x,xs] singleton x ∘ xs|
```

Figure 6: Constructors of collection type

Since collections are no simple data type it is more difficult to reason about them, more precisely to reason about a general collection through its recursive structure. This results in the need of induction on collections. The induction in MMT needs a formalized framework typically defined through axioms. The axiom used for collections is *inductive_proof*. Not all proofs are about the effect of a function on the collection but on a more general level. In that case it is sufficient to know the properties of the collection type. This is given though the axiom cons_or_neutral.

The collection type should be viewed as an interface for the recursive wrapping data types in functional programming. With this abstract type real types can be created. These sub types of collection have their own specification theories defining additional mathematical concepts of their type. For lists it is ListSpec with EndoMonoid properties. The instantiation is then realized using the *structure* keyword in MMT. The resulting object is called *list* for the *List* type. The effect of this is the internal separation of the types. Otherwise the types would have the same structural components and thus would be difficult to use simultaneously in one theory. The benefit is the possibility of renaming the already defined components. Every element included in the specification of the type can be called upon. For example in *List* the function *length* is not renamed and can be used by *list/length*. The already instantiated types are List, MultiSet, FiniteSet, Binarytree, and Option. These types have common functions already defined in collection but also unique methods defined in their specific theories.

List For the List type a mathematical framework of EndoMonoid is used. It adds the properties of a semigroup to the framework provided by collection. This adds the associative rule. The resulting type is a good representation of a typical list. The *op* is renamed to *concat* and the *appliedtype* to *List*. The neutral element is the empty list and named *nil*. The basic functionality is provided by this theory, but additional functions are formalized in the theories ListOperator and ListIterator, see figure 7. The two most common functions on lists are head and tail, where head gives the first element and tail the remaining list after the first element. These functions can not be formalized generally for all collections because some implemented types have different assumptions. An example are sets which have no order.

List_operators	List_Iterators
hd	list_ map
tl	foldl
drop	foldr
add_last	l_exists
rev	l_all
list_match	filter
	zip
	unzip

Figure 7: List functions

Multiset By adding the commutative law though the EndoCommutitative theory to the List framework the mathematical foundation for the Multiset is given. Through this added law the relative position of every element becomes irrelevant. Because of this distinction there are differences in methods applicable to the data types. The Multiset type is formalized with more of a mathematical than a programming perspective. The functions *hd* and *tl* which are formalized in List could also be formalized in Multiset but aren't because they do not fit in the properties of the type. The first element cannot be taken out of a set because there is no first element. An overview of functions formalized for Multiset is shown in figure 8.

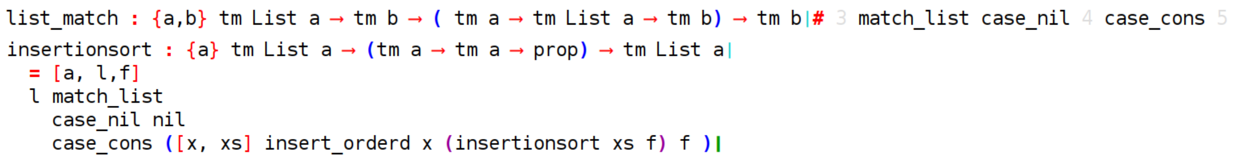
Multiset_operators
equal
union
intersect
distinct_size
occur
remove
removeall

Figure 8: MultiSet functions

FiniteSet The FinitSet is created using the mathematical framework of Multiset and adding the law of idempotent. The theory added is EndoIdempotent. With this new addition the data type restricts every element to only occur once. The functions formalized are very similar to the ones for Multiset.

Binarytree : The binary tree is a common data structure in programming languages for easily finding an element in a large data structure. The data structure could be formalized in a new form of collection. But it was tried to represent it in the already defined one because it would be beneficial to have one base class and not many different ones. In the binary tree the leaf can easily be represented by *singleton*, which creates a tree out of a single object. The connection of two trees would normally be done by a node containing the left and right branch and a single value. The function normally used for this would be the cons function, but it accepts only one complex data type as argument and therefore cannot represent a tree. Thus, the option which is used is the *op* function connecting two complex types. However, with this approach there would be no values in the nodes. The cons operator thus has the functionality as shortcut for connecting a singleton and a tree branch. Even though the neutral element is present in the theory it is irrelevant. In summary the resulting tree allows for data only in the leaves, and the theory contains functions which have no real value. The resulting structure would probably never be used because it fails storing data efficiently and it is difficult to find the right element. However, with a slight modification of *Collection* a real tree could be realized. The cons operator could be modified to allow an additional complex data type. Also the op function should be replaced as it needs to allow nodes with a value.

In functional programming languages it is common to have a form of pattern matching especially for recursive data structures like collection types. Such a general tool can not be simply formalized in LF, since it would not be possible to match against a general form of term. Also it would be difficult to formalize something for an arbitrary type. Therefore only a simplistic version is provided: for every type that requires pattern matching it needs to be implemented separately. The concept can be seen in the example *list_match* in figure 9. *list_match* matches on nil (the neutral element) and cons. In the nil case only a return value is needed because that case would get always the empty list as argument. In the cons case a function is needed, which gets as first argument the value of the cons(x in this example) and as second the remaining list (xs). Most programming languages allow for an arbitrary matching, but that is only syntactic sugar and this form is equally powerful.



```
list_match : {a,b} tm List a → tm b → ( tm a → tm List a → tm b) → tm b|# 3 match_list case_nil 4 case_cons 5
insertionsort : {a} tm List a → (tm a → tm a → prop) → tm List a|
  = [a, l,f]
  l match_list
    case_nil nil
    case_cons ([x, xs] insert_orderd x (insertionsort xs f) f )|
```

Figure 9: Example list_match

The diagram in figure 10 displays the theories relevant for the collection types. The green nodes contain the mathematical framework in which the collection types are formalized. These were already present in Latin. The light blue theories are the specification for the collection and the instantiated types. The orange-colored nodes are the instantiated types. The type specific functions are contained in the dark blue nodes.

5 Monads

In this library some monad related theories are formalized. The structure and types formalized were inspierd by the Haskell type class hierarchy [11]. On one hand there are the general theories Functor, Applicative, and Monad which are the building blocks the Monad types are built form, already present in Latin. On the other hand there are specialized theories describing real types like ListMonad, OptionMonad, FailureMonad, StateMonad, and IOMonad. These specialized monads can be further divided in those only proving the monadic properties for a type and those needing the monad to function.

5.1 Related Theories

EndoFunctor This is the simplest form of classification used in these monadic forms. The class describes the application of a function on a container. In Latin the

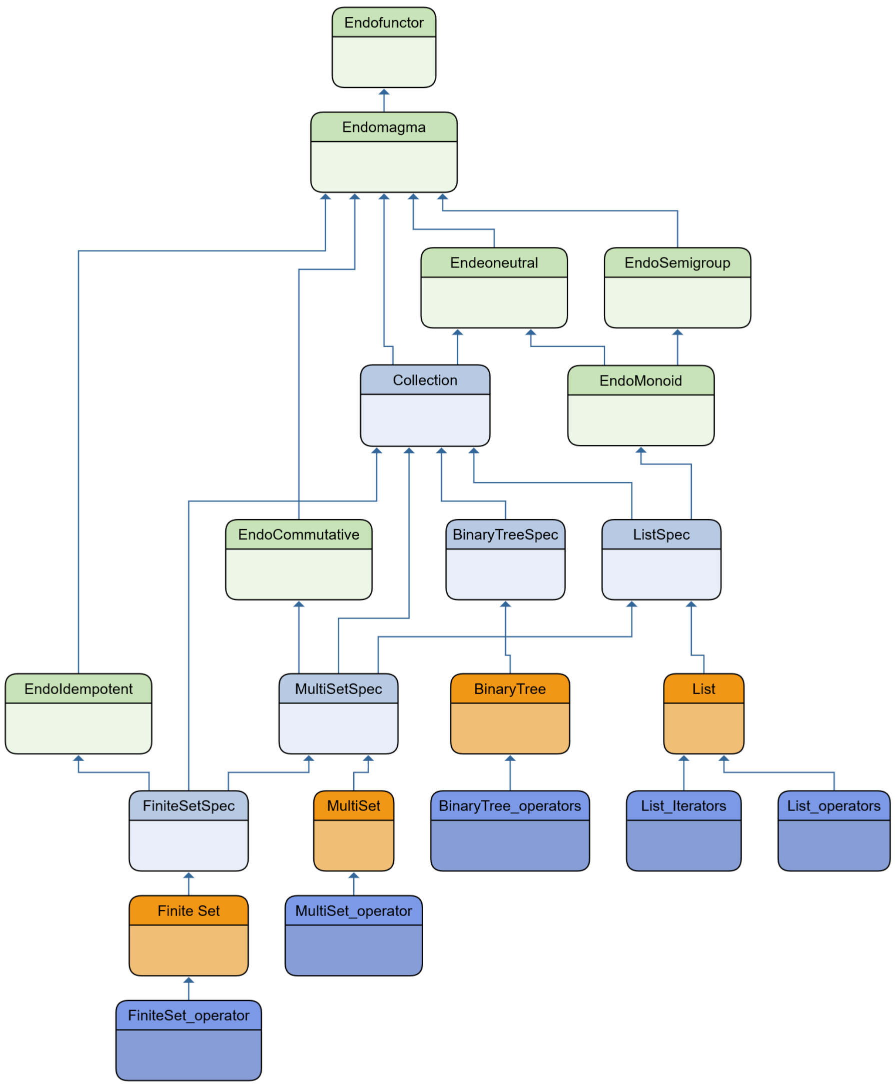
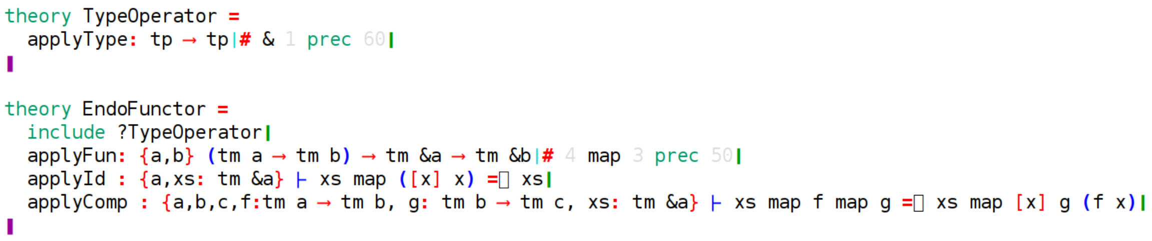


Figure 10: The collection framework

container used for EndoFunctor is the applytype. The description of the function is the applyFun. The function can also be described as map. The exact formal definition is shown in figure 11. This functor is the base for the other classifications.

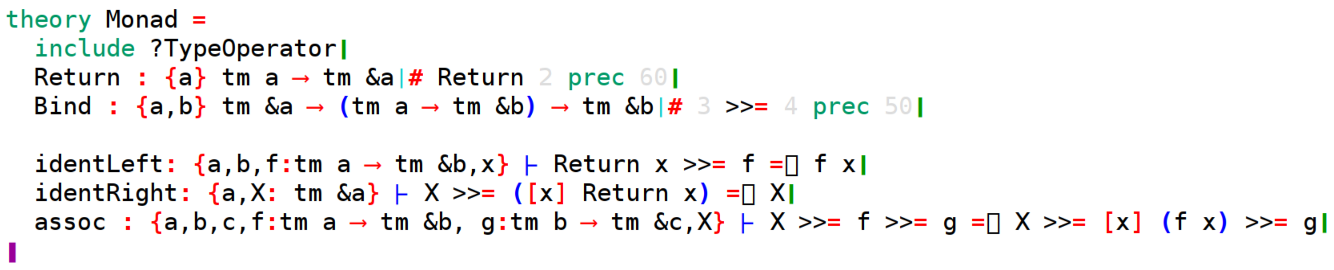
Monad The monad formalization contains the base type operator of applyType, which is the type the monad operates on, and the functions return and bind. Return is the function creating the monad from an object. The bind function describes how functions operate on the monad itself. There are also the axioms all monads need to fulfill: identRight, identLeft, and associative. See figure**??** for the implementation.

Applicative The Latin library contains the Applicative theory shown in figure 13 The Applicative type is a functor weaker than Monad. It allows for a more widespread use. The idea of Applicative is that *pure* wraps pure computation into the part of the environment which is pure. *splat* is then the function applying



```
theory TypeOperator =
  applyType: tp → tp❘# & 1 prec 60❙
❚

theory EndoFunctor =
  include ?TypeOperator❙
  applyFun: {a,b} (tm a → tm b) → tm &a → tm &b❘# 4 map 3 prec 50❙
  applyId : {a,xs: tm &a} ⊦ xs map ([x] x) =▯ xs❙
  applyComp : {a,b,c,f:tm a → tm b, g: tm b → tm c, xs: tm &a} ⊦ xs map f map g =▯ xs map [x] g (f x)❙
❚
```

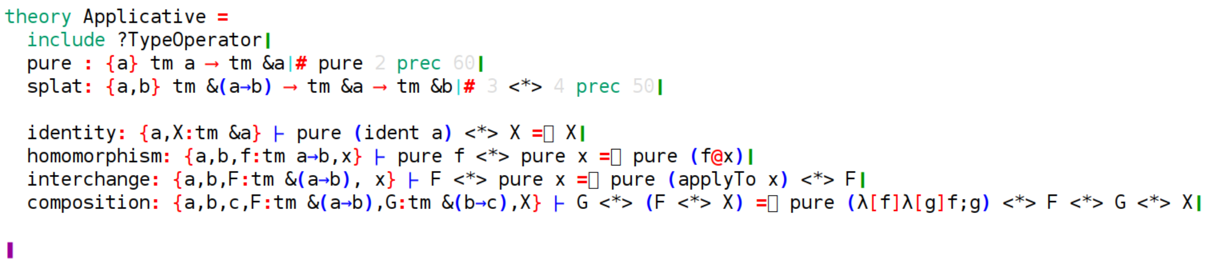
Figure 11: applyType and Endofunctor



```
theory Monad =
  include ?TypeOperator❙
  Return : {a} tm a → tm &a❘# Return 2 prec 60❙
  Bind : {a,b} tm &a → (tm a → tm &b) → tm &b❘# 3 >>= 4 prec 50❙

  identLeft: {a,b,f:tm a → tm &b,x} ⊦ Return x >>= f =▯ f x❙
  identRight: {a,X: tm &a} ⊦ X >>= ([x] Return x) =▯ X❙
  assoc : {a,b,c,f:tm a → tm &b, g:tm b → tm &c,X} ⊦ X >>= f >>= g =▯ X >>= [x] (f x) >>= g❙
❚
```

Figure 12: Monad theory

functions to this wrapped type. An applicative type needs to fulfill the axioms identity, homomorphism, interchange, and composition. [4].



```
theory Applicative =
  include ?TypeOperator❙
  pure : {a} tm a → tm &a❘# pure 2 prec 60❙
  splat: {a,b} tm &(a→b) → tm &a → tm &b❘# 3 <*> 4 prec 50❙

  identity: {a,X:tm &a} ⊦ pure (ident a) <*> X =▯ X❙
  homomorphism: {a,b,f:tm a→b,x} ⊦ pure f <*> pure x =▯ pure (f@x)❙
  interchange: {a,b,F:tm &(a→b), x} ⊦ F <*> pure x =▯ pure (applyTo x) <*> F❙
  composition: {a,b,c,F:tm &(a→b),G:tm &(b→c),X} ⊦ G <*> (F <*> X) =▯ pure (λ[f]λ[g]f;g) <*> F <*> G <*> X❙

❚
```

Figure 13: Applicative theory

5.2 Monadic Types

Collection The formalized collection types can be introduced to the Monadic type hierarchy. This is done by defining the monad of their type. Therefor the applyType as well as the return and bind functions need to be defined. This is shown for List in 14. The indentLeft axiom proof is simply done by the *trefl* tactic. The other two should be possible but are omitted.

FailureMonad In many programming languages a form of exception throwing and handling exists. The exceptions in prominent programming languages provide large amounts of information on the error that has occurred. For formalization such would be relatively difficult and time consuming if possible at all. A simplified version of an exception can be a string containing an error message provided through the return value. The FailureMonad operates on the combined data type of String and a type a. The monad provides a functionality to use the type simply as if it was type a. This is done through the functions return and bind. Return

```
view List_Monad : ?Monad → ?Lists =
  applyType = [x] List x ❘
  Return   = [a,x: tm a] list/singleton x  ❘
  Bind = [a,b,X: tm List a,F: tm a → tm List b] list/map2 F X  ❘
  identLeft = [a,b,f,x] trefl ❘
❚
```

Figure 14: ListMonad view

creates the monad out of an object of type a, while bind executes a function having a value of type a as the only argument. Though rightinjection on the union of a and string an error message can be returned. This monad would also allow for implementation of a catch function in the future.

StateMonad Through the properties of pure functions, functional programming languages don't have a global state. However, there are some algorithms which can be implemented much simpler if a form of state exists. The language could introduce mutable variables which are in conflict with the pure functional programming paradigm. A different way is to add the state as an additional argument for functions. The state should be passed on to every function call and be returned with the result. The StateMonad represents this way of introducing states to functional programming. It is realized by combining the result type with the state into a tuple. The different forms of states can be accommodated by different formalizations of the tuple. Bind and return are only functions operating on the return value, but not on the state value. There are formalized functions providing options to manipulate the state value.

IOMonad IO operations are an important factor of programming as they allow for interaction with the real world. These operations are essential if a program written in this language needs to be executed. Do to their nature, functional programming languages have difficulties with such functions. The real world interaction would eliminate the deterministic nature of the program. This is the result of the world being state dependent. A solution for including this impurity in the functional language is through an IOMonad. This monad hides the impurities as they are handled separately from the normal use.

The resulting type diagram is displayed in figure 15. The blue nodes are framework theories, the yellow ones are the "real monad types" and the green ones are the converted types. The yellow and green theories were added by this work.

6 Towards turing completeness

The previous chapters described types used in programming languages as well as functions operating on these types. But a programming language does not only need types. Some essential functionality programming languages need to be fully functional are discussed in this chapter. The two main topics are recursion and

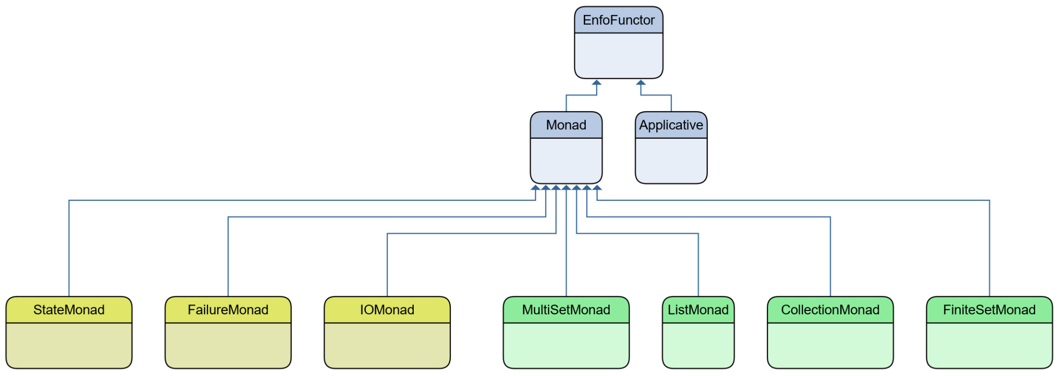


Figure 15: Diagram of the monadic theories

exception handling. Additionally, some mutable elements are discussed and an example is presented of how programs could be implemented in Latin.

6.1 Recursion

The most important programming technique in functional programming is recursion. The ability of a function to call itself allows for decomposing a difficult task into its simplest parts and incrementally solving the overall task. Because functional programming does not have loops, recursion is the only solution for problems with an arbitrary number of steps. Unfortunately, currently MMT does not support the use of a function in its declaration and thus eliminates the use of the most common form of recursion. A version of recursion which could be formalized in MMT is an anonymous recursion operator. This solution would formalize an operator applying a function recursively as shown in figure 16. This operator could be defined using axioms. Although this form is possible in the current MMT version it has some problems. With this approach, a function using normal recursion can easily be formalized, but mutual recursion schemes would cause problems. Also recursive functions having more than one argument could not be formalized using this definition. With these problems and the less convenient use of the anonymous recursion operator (figure 17) compared to conventional implementation of recursion, changes to MMT are considered the better solution. An approach which would allow that is to add a new concept of *program* in parallel to the existing concept of *theory*. The efficiency of the existing parser, resulting from prohibiting recursive usage of functions, would remain, while still enabling recursion for formalization of functional programming related algorithms. Existing theories would not need to be changed.

```
recfun: {b,c} (tm a → b → tm b → c ) → tm b → c ❘# %n 1 2 3 ❙
```

Figure 16: Possible formalization of the anonymous recursion operator

```
factorial = recfun nat nat ([f][n] if n == 0 then 1 else n*f(n-1))❙
```

Figure 17: Example of recursion using a recursion operator

6.2 Exception handling

In computer science there are three common practices of handling errors. The simplest one is simply leaving it to the programmer. Here, the programmer has the task of guaranteeing only to use the functions with the right terms, i.e. ensuring that no errors actually occur. In other words, errors are not prevented. Thus the implementation of the base library is simpler at the expense of more effort for the programmer. An alternative is the options data type. It is the form used in most functional programming languages. In simple terms, it is a wrapper around the result. In case a function was successful, the option contains the result, otherwise it is empty. The advantage of this type is that it is clear afterwards if the return value is a correct answer or just produced by some error that occurred during the execution. The disadvantage of this approach is that it does not provide any sort of information about the error occurred. The data type of Options needs to be present for the use in that program.
The most advanced form of error handling is throwing an exception. This stops the function when an error occurs and returns an error message to the caller of the function. This form of error handling is the most advanced way but also needs the most additional functionality, because a way of passing error messages needs to be added and the catching of these must be introduced. This would be difficult especially in pure formalization. This work decided to use the first option for the formalization of the library. The argument against the second option of error handling was the position at which it was formalized. Since option is a sub type of collection it wouldn't be usable in the formalized functions. It would be possible to add the type additionally at a lower level but this would lead either to two separate types or options not formalized as a subtype of collection.

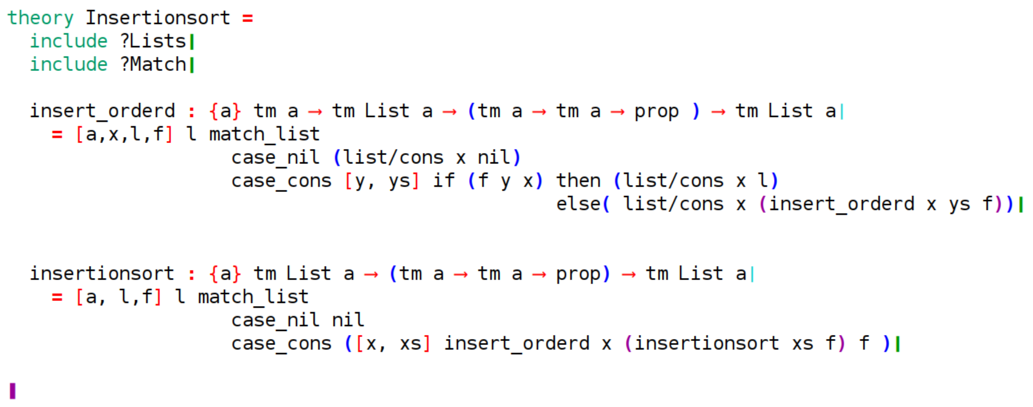
6.3 Mutable

Many programming languages designed in one paradigm add functionality from another paradigm. An example are the functions operating on mutable variables in Scala. This concept describes the need for stateful functions in a pure functional language. This library is not formalized for one language but to allow representation of many languages. In addition it is intended to complement the library with an formalization of an object oriented programming language. Because of this some formalization for mutable variables is added as well as functions operating on them.
For formalization purposes the unit type is needed, which provides a solution to the problem of no return type. Some functions don't have return values but change a state variable. Examples of those functions are IO operations, variable declarations, value assignment, and structural elements like loops. Even though the types of these functions can be formalized, the underlying functionality can not. Because of this, rewriting is no option as variant for execution. But there is a template in MMT for the role execution. This would map onto a function written in Scala realizing the functionality. The theories related to mutable types can be found in the control_flow and Datatypes files in the programming folder of Latin.

6.4 Programs in MMT

In order to provide examples for how to formalize algorithms and to prove that the presented methodology is useful, two well known sorting methods were formalized using the library developed in this work.

Insertion sort Insertion sort is the algorithm sorting a list through sorted insertion of every element into a new list. The function utilizes a helper function which performs the sorted insertion. The algorithm iterates over every element through pattern matching. In the case of the empty list an empty list is returned. In the case of cons the function does the recursive step with the tail. The variable of cons is then inserted into the result of the recursive step.
The sorted insertion is done through pattern matching on the list. In case of the empty list a new list is created with the element to be inserted as sole element. In case of cons the given weighting function decides which element is the matching one. If it is the place of the inserted object it is append to the list otherwise a recursive step is made and append to the head of the returned list.



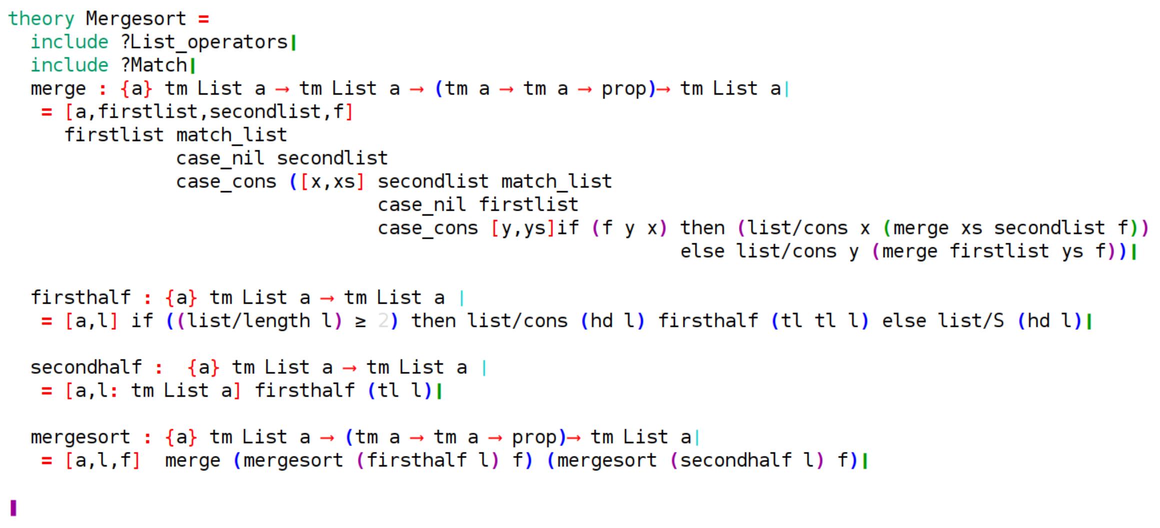
```
theory Insertionsort =
  include ?Lists❙
  include ?Match❙

  insert_orderd : {a} tm a → tm List a → (tm a → tm a → prop ) → tm List a❘
    = [a,x,l,f] l match_list
                  case_nil (list/cons x nil)
                  case_cons [y, ys] if (f y x) then (list/cons x l)
                                                 else( list/cons x (insert_orderd x ys f))❙


  insertionsort : {a} tm List a → (tm a → tm a → prop) → tm List a❘
    = [a, l,f] l match_list
                  case_nil nil
                  case_cons ([x, xs] insert_orderd x (insertionsort xs f) f )❙

❚
```

Figure 18: InsertionSort

Merge sort The algorithm receives two lists (x and y) and a sorting function (f). The sorting algorithm consists of two main functions: splitting and merging. The main method mergesort splits the list into two equal sized partial lists and continues by recursion until all lists only contain one element. After the split, both parts of the list are merged in a sorted fashion through the function merge. This function uses pattern matching on the first list and then on the second. In the simplest case one list is already empty, leading to returning the other list. Otherwise both lists contain a head element (x for the fist list and y for the second). Then the sorting function determines the new head. The tail is calculated by a recursive call with the remaining lists.

7 Conclusion and future work

In this work, the foundation has been created for formalizing a functional programming language in MMT. This includes the base types of Boolean, Numbers,



```
theory Mergesort =
  include ?List_operators❙
  include ?Match❙
  merge : {a} tm List a → tm List a → (tm a → tm a → prop)→ tm List a❘
   = [a,firstlist,secondlist,f]
     firstlist match_list
               case_nil secondlist
               case_cons ([x,xs] secondlist match_list
                                 case_nil firstlist
                                 case_cons [y,ys]if (f y x) then (list/cons x (merge xs secondlist f))
                                                          else list/cons y (merge firstlist ys f))❙

  firsthalf : {a} tm List a → tm List a ❘
   = [a,l] if ((list/length l) ≥ 2) then list/cons (hd l) firsthalf (tl tl l) else list/S (hd l)❙

  secondhalf :  {a} tm List a → tm List a ❘
   = [a,l: tm List a] firsthalf (tl l)❙

  mergesort : {a} tm List a → (tm a → tm a → prop)→ tm List a❘
   = [a,l,f]  merge (mergesort (firsthalf l) f) (mergesort (secondhalf l) f)❙

❚
```

Figure 19: MergeSort

and Strings as well as the more complex collection types. Additionally some basic monad types where introduced. Two potential realizations of recursion were described with a preference for adding new syntax to MMT. The different ways of exception handling were introduced with formalization of the option type for the user and using no error handling in the formalized functions. The resulting framework shows that programs written in a functional language could be formalized in Latin.
The added formalization produced for this work can be found in https://gl.kwarc.info/supervision/bloecher_moritz/-/tree/main/%20bachelorarbeit.

Future work In future a version of recursion needs to be introduced to the framework. Without such a tool most programs cant be formalized. Two possible variants were discussed.
A different aspect which should be regarded is a better idea for sub types as mentioned for the case of numbers. The introduced formalization is usable in programming but has some problems. With a better sub typing system these problems could be eliminated.
Another interesting area is the execution of programs. The current formalization does not possess a real processing ability. The formalization would need IO functions coupled with a console and solutions for those functions which can't be formalized by axioms suited for role Simplify.
Another task would be to try to represent existing programming languages in Latin. One task corresponding to this is to expand the formalized framework. This can be done in different ways either by introducing new representation of the present types or by adding different language designs like object oriented programming.

References

[1] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *International Summer School on Applied Semantics*, pages 42–122. Springer, 2000.

[2] Emden R Gansner and John H Reppy. *The Standard ML basis library*. Cambridge University Press, 2004.

[3] Paul Hudak and Joseph H Fasel. A gentle introduction to haskell. *ACM Sigplan Notices*, 27(5):1–52, 1992.

[4] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

[5] F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

[6] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.

[7] John Reppy. The standard ml basis library. In *https://smlfamily.github.io/Basis/index.html*, 2002.

[8] The KWARC research group. Latin2 github repository. In *https://gl.mathhub.info/MMT/LATIN*. Dept. of Computer Sciences, The KWARC research group Friedrich Alexander University, 2020.

[9] ”The KWARC research group”. Latin logic atlas und integrator. In *https://kwarc.info/projects/latin/*, 2022.

[10] Twelf team. Lf. In *http://twelf.org/wiki/LF*, 2021.

[11] Brent Yorgey. Typeclassopedia. In *https://wiki.haskell.org/Typeclassopedia*, 2021.